

A portable and efficient implementation of global constraints: the `tree` constraint case

Guillaume Richaud, Xavier Lorca, and Narendra Jussien

École des Mines de Nantes, LINNA FRE CNRS 2729, FR – 44307 Nantes Cedex 3
{guillaume.richaud,xavier.lorca,narendra.jussien}@emn.fr

Abstract. Global constraints represent invaluable modeling tools for Constraint Programming (CP). Efficiently solving recurrent subproblems is a key point for CP successes. However, global constraints mainly remain strongly attached to a given constraint solver. Indeed, they heavily rely on internal mechanisms in order to be as efficient as possible. In this paper, we emphasize the interest of decoupling global constraint implementations from the underlying solver. We show, on a `tree` constraint, that even more decoupling it by providing fully dynamic algorithms enhances efficiency and, which is much more important, allow an efficient portability of the constraint. We illustrate this for the *Choco* and *Gecode* solvers.

1 Introduction

Constraint Programming (CP) is an ever evolving field whose aim it is to solve combinatorial problems in a declarative and flexible paradigm. At the heart of a constraint program is a constraint satisfaction problem (CSP) which is defined by a set $\mathcal{V} = \{v_1, \dots, v_n\}$ of variables (in the mathematical sense), a set $\mathcal{D} = \{dom(v_1), \dots, dom(v_n)\}$ of domains which represent the set of possible values that each variable can take, and a set \mathcal{C} of constraints (relations) upon subsets of variables. A solution for a CSP is a variable assignment (a value for each variable) that simultaneously satisfies the constraints of the problem. A constraint solver is meant to look for such a solution. End-users of constraint programming only need to enunciate the variables and the constraints of their problem.

In this context, *global constraints* represent invaluable modeling tools for the CP field. Indeed, global constraints represent compact solutions and solving algorithms for recurrent subproblems in CSP. They are used as classical constraints and usually encompass a set of constraints defined upon a large set of variables. For example, the well-known `alldifferent` constraint [11] is used to replace a clique of difference constraints. Global constraints offer a more precise and more efficient view of the subproblem they are defined upon. They actively use the underlying structure to provide efficient filtering algorithms. Indeed, the explicit knowledge of this structure leads to an improved propagation-search technique by avoiding repeatedly discovering the same inconsistencies (this phenomenon is called *thrashing*). As expected, global constraints usually imply a higher worst-case time complexity for the filtering algorithm w.r.t. the original set of constraints. However, this overhead is most often largely compensated by the filtering power achieved by the global constraint. Actually, there exists a trade-off between efficiency (*i.e.* running time) and effectiveness (*i.e.* filtering power).

Powerful global constraints are generally attached to one solver: `cycle`, `diffn`, `cumulative` with `chip` [1], `tree` [2] with `choco`, `standard deviation` [12] with `Ilog`. This is probably due to the fact that implementing a global constraint can be highly solver-dependent. Indeed, global constraint implementations usually involve both internal data structure and solver-related structures. The latter are most often backtrackable structures offered by the solver to be used by the constraint to ease the implementation. Not all solvers provide the same backtrackable structure leading to solver-dependent constraint implementations. This can lead to less efficient constraints from one solver to another. This is for example the case with the `alldifferent` constraint (one of the rare ones that made it through several solvers). The efficiency of the constraints is not the same w.r.t. the services provided by the host solver.

In this paper, we would like to investigate these aspects of global constraint implementations. More precisely, we are interested in pointing out that once a global constraint has been defined it needs quite a fine tuning to take advantage of the underlying constraint solver. Hence, we would like to address the point of being able to provide both flexible (in the sense of easing addition or removal of filtering algorithms) and portable (in the sense of being able to adapt the constraint to another solver) implementations of global constraints.

Our test case throughout the paper will be a graph partitioning constraint introduced in [2], the `tree` constraint. We will show that implementing fully dynamic filtering algorithms (*i.e.* not relying upon backtracking and managing their own data structure) transforms the constraint as a plugin for the solver (see Figure 1).

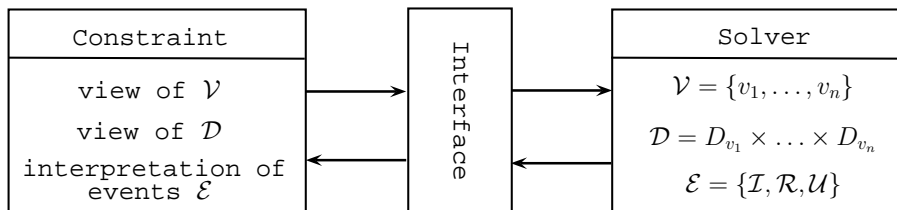


Fig. 1: A *pluggable constraint* defined according to three kinds of data: a view of the variables \mathcal{V} that define the problem, a view of the domain of each variable \mathcal{D} and, an interpretation of the events that can occur on the variables. These events could be the *instantiation* (\mathcal{I}) of a variable to a value, the *removals* (\mathcal{R}) of a value in the domain of a variable and, the *update* (\mathcal{U}) of the lower bound or the upper bound of a variable’s domain.

The paper is organized as follows: we first provide a quick description of the `tree` constraint and the current implementation provided within the `choco` constraint solver (<http://choco-solver.net>). Next, we show how the filtering rules used in this constraint can be implemented with fully dynamic algorithms, leading to a pluggable `tree` constraint. We show that this constraint is pluggable by integrating it into `gencode` (<http://gencode.org/>). Moreover, we show that the pluggable version is more efficient with `choco` than the fully integrated version and show that the

gecode version can be made quite as efficient despite the event management system of gecode's java interface.

2 Description of the `tree` constraint

The `tree` constraint partitions a given directed graph (digraph for short) into a forest of node-disjoint trees. More precisely, the digraph is partitioned into a set of node-disjoint anti-arborescences¹. `tree` is a useful constraint that can be used for modeling various graph-related problems like, for example, surpertime phylogenetic problems [3, 6], ordered disjoint path problems [3, 10], or mission planning problems [7].

The constraint has the form `tree(NTREE, VER)`, where `NTREE` is a domain variable² specifying the number of trees in the forest (`MINTREE` and `MAXTREE` respectively denote the minimum and maximum values of $\text{dom}(\text{NTREE})$) and, `VER` is the collection of n nodes `VER[1], ..., VER[n]` of the given digraph. Each node $v_i = \text{VER}[i]$ has the following attributes, which complete the description of the digraph:

- `L` is a unique integer in $[1, n]$. It can be interpreted as the *label* of v_i .
- `F` is a domain variable whose domain consists of elements (node labels) of $[1, n]$. It can be interpreted as the *unique successor* (or *father*) of v_i .

When speaking of global constraints, it is often convenient to reason about a digraph that models the constraint rather than directly about the constraint. We model the extended `tree` constraint by the digraph \mathcal{G} in which the nodes represent the elements of `VER` and the arcs represent the successor relation between them. Formally, \mathcal{G} is defined as follows:

Definition 1 (Associated digraph to a `tree` constraint). *The associated digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of a `tree(NTREE, VER)` constraint is defined by $\mathcal{V} = \{v_i \mid i \in [1, n]\}$ and $\mathcal{E} = \{(v_i, v_j) \mid j \in \text{dom}(\text{VER}[i].F)\}$.*

A `tree(NTREE, VER)` constraint specifies that its associated digraph \mathcal{G} should be a forest of `NTREE` trees, formally:

Definition 2 (Solution of a `tree` constraint). *A ground instance of a `tree(NTREE, VER)` constraint is said to be a solution if and only if:*

- $\forall i \in [1, n] : \text{VER}[i].L = i$.
- *The associated digraph \mathcal{G} consists of `NTREE` connected components.*
- *Each connected component of \mathcal{G} has no circuit involving more than one node (notice that each component contains exactly one node that has a self-loop and that corresponds to the root of the tree).*

We recall some definitions and notations regarding the digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ associated with a `tree` constraint, as well as a lower and upper bound on the number of trees needed for partitioning \mathcal{G} . These notions are introduced in the original version of [2]:

¹ A digraph \mathcal{A} is an *anti-arborescence* with *anti-root* r iff for each node v in \mathcal{A} there is a path from v to r and the underlying undirected graph of \mathcal{A} is a tree.

² A *domain variable* V is a variable ranging over a finite set of integers denoted by $\text{dom}(V)$; $\text{min}(V)$ and $\text{max}(V)$, respectively, denote the minimum and maximum values of $\text{dom}(V)$.

Definition 3 (Reduced graph). To each instance of a `tree`(NTREE, VER) constraint we associate the reduced digraph \mathcal{G}_r derived from \mathcal{G} in the following way: to each strongly connected component of \mathcal{G} we associate a vertex of \mathcal{G}_r ; to each arc of \mathcal{G} that connects different strongly connected components corresponds an arc in \mathcal{G}_r .

Notations 1 (Sink component) A strongly connected component of \mathcal{G} that corresponds to a sink of \mathcal{G}_r is called a sink component.

Notations 2 (Potential root & loop) A node v of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that $(v, v) \in \mathcal{E}$ is called a potential root. The arc (v, v) is called a loop.

Notations 3 (Door) A node u of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a door of the strongly connected component associated with u iff there exists $(u, v) \in \mathcal{E}$ such that u and v do not belong to the same strongly connected component of \mathcal{G} .

Definition 4 (Dominator [9]). Given a digraph \mathcal{G} and two distinct nodes i, j of \mathcal{G} such that there is at least one path from i to j , a node d is a dominator of j with respect to i iff there is no path from i to j in $\mathcal{G} \setminus \{d\}$. The set of dominators of j with respect to i is denoted by $DOM_{(\mathcal{G}, i)}(j)$.

We are now in position to detail how the initial version [2] of the `tree` constraint is effectively implemented in Choco. Next, we propose a fully dynamic version of this constraint allowing a new design of its implementation that exploits the incrementality in order to avoid using backtrackable data structures dedicated to the solver. A nice property of such an implementation is that the `tree` constraint becomes a plugin for any solver.

3 Implementation of `tree` constraint

3.1 An *ad-hoc* version for *choco*

Several constraint solvers are available but not two of them provide the exact same set of global constraints. When facing a constraint satisfaction problem, the choice of the constraint solver therefore highly depends on the desired constraints and their efficiency. Moreover, quite often only some part of the problem is efficiently handled and one need to either develop or simulate a global constraint in order to solve the problem. Thus, when implementing a global constraint two situations arise: in the first one, the constraint solver has already been chosen and therefore the provided data structures must be used; in the second one, one can choose the most promising solver for the constraint, i.e. a solver which proposes the most adapted data structures to the filtering algorithms to be implemented. In the `tree` constraint case, the *choco* constraint solver was selected. Indeed, it allows a fine-grained management of events³ occurring on variables. Each variable associated to a problem knows the constraints that involve

³ *Choco* distinguishes three main kinds of events: instantiation of a variable, removal of a value in the domain of a variable, update of the lower bound of the domain of a variable and, update of the upper bound of the domain of a variable.

it, and symmetrically, each constraint knows the subset of variables that it is posted upon. Then, for a given constraint, distinct treatments can be done for each kind of event occurring on variables involved in the constraint. This leads to the fact that a fix point can easily be reached because the constraint may specify if it has to be awoken by an event produced by itself. The main benefit of such a property is that for a given n -ary constraint, the implementation of its filtering algorithm can be triggered for each kind of event that occurs on the variable domains involved in the constraint.

We now provide the skeleton of the `tree` constraint such as it is implemented in the *choco* solver. The details of each kind of propagation can be seen in [2]. Notice that, w.l.o.g., the notion of *strong articulation points* is generalized to the notion of *dominators* [9]. Obviously, this generalization does not change any filtering algorithm of the initial paper except the computation of dominators in the digraph \mathcal{G} associated with the `tree` constraint.

Initial awake of the `tree` constraint:

- Compute MINTREE and MAXTREE.
- If there is at least one solution satisfying the constraint then, do propagation related to the constraint:
 1. Update NTREE according to MINTREE and MAXTREE.
 2. Propagate according to the dominator nodes of \mathcal{G} .
 3. Propagate according to the doors and the potential roots of \mathcal{G} .
 4. Propagate according to the values of $\max(\text{NTREE})$ and $\min(\text{NTREE})$.

Each time an event occurs on a domain variable involved in the `tree` constraint do:

- If this event occurs on a domain variable modeling NTREE then:
 1. Update NTREE according to MINTREE and MAXTREE.
 2. Propagate according to $\max(\text{NTREE})$ and $\min(\text{NTREE})$.
- If this event occurs on a domain variable modeling a node of \mathcal{G} do:
 1. Update NTREE according to MINTREE and MAXTREE.
 2. Propagate according to new dominators of \mathcal{G} .
 3. Propagate according to the doors and the potential roots of \mathcal{G} .

The *choco* constraint solver is based on a *trailing*⁴ [13] approach to record a decision (e.g., instantiation of variables, removals of values in the domains, etc) and its effects on the data structures involved in the constraint. In our purpose, these effects consist in modifications of the graph structure, for example, if an arc (i, j) is removed from \mathcal{G} (i.e., $j \notin \text{dom}(\text{VER}[i].F)$) then, this removal may:

1. decrease the number of potential roots (see Notation 2), if $i = j$. This leads to an update of MAXTREE.
2. increase the number of sink components (see Definition 1). This leads to an update of MINTREE.

⁴ A *trailing* approach records for each events modifying a data structure, the necessary information to undo its effect.

3. increase the number of strongly connected components (scc) in \mathcal{G} . This leads first to increase the number of doors (see Notation 3) contained in \mathcal{G} , and second, to change the reduced digraph \mathcal{G}_r associated with \mathcal{G} (see Definition 3).
4. create new dominator nodes in \mathcal{G} (see Definition 4).

Thus, in order to record the necessary information, the Choco solver proposes some *backtrackable* data structures using storable integers, storable booleans and storable bit-sets. The original `tree` constraint uses these backtrackable data structures in order to dynamically record and restore some graph properties like strongly connected components and dominator nodes of the digraph \mathcal{G} .

However, state-of-the-art graph algorithms propose several fully dynamic algorithms [5] maintaining the graph properties involved in the `tree` constraint like strongly connected components and transitive closure. Then, two straightforward issues are: is it really necessary to use backtrackable data structures when fully dynamic algorithms exist? If no backtrackable data structures are finally used (i.e. it is not necessary to trail the variation of the data structures involved in the constraint) what are the exact relations between the constraint and the solver?

One interesting point here is that when being able to provide *solver-independent* global constraints, other perspectives are open: such a constraint could be plugged in other problem solvers. For example, a good candidate could be COMET, a local search development environment [8], another interesting one would be PaLM [4], the explanation-based extension of the Choco solver.

3.2 A pluggable `tree` constraint

On the one hand, one can summarize the bottleneck of the complexity of a `tree` constraint to repeatedly maintaining several graph properties (strongly connected components (scc), transitive closure, dominator nodes) related to the digraph \mathcal{G} associated with \mathcal{G} between two search steps. For each property, several filtering algorithms are proposed in order to remove inconsistent arcs during the propagation step. On the other hand, the propagation-search techniques only consist in selecting and removing values in variable domains, or restoring values in variables domains. In the context of a `tree` constraint, this technique modifies the `NTREE` variable as well as the set of arcs involved in the digraph \mathcal{G} associated with the constraint.

Basically, a new approach implementing such a constraint can be decomposed in the following way (Figure 2):

1. The **Graph module** is based on a generic fully incremental data structure modeling a digraph \mathcal{G} and its associated properties (i.e., scc's and transitive closure). This module contains primitives updating the data structure according to arc removals and restorations. These primitives basically compute each property on a necessary partial graph⁵ of the original digraph \mathcal{G} .
2. The **Constraint module** proposes the filtering algorithms (based on the properties maintained by the graph module) that remove arcs of \mathcal{G} inconsistent with the `tree` constraint.

⁵ Given a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a *partial graph* \mathcal{G}' of \mathcal{G} is defined by $(\mathcal{V}' \subseteq \mathcal{V}, \{(i, j) \in \mathcal{E} \mid i, j \in \mathcal{V}'\})$.

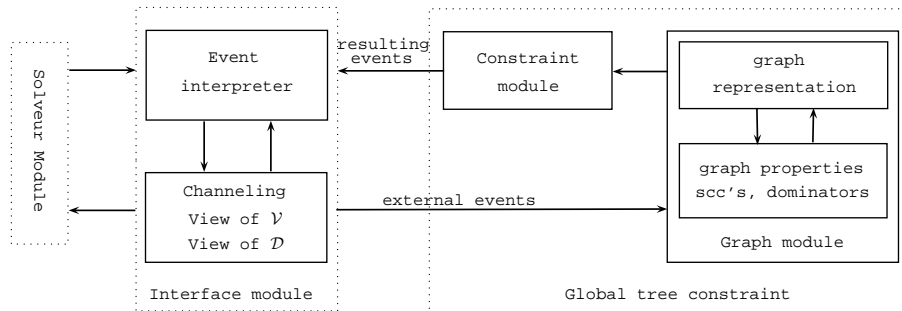


Fig. 2: General implementation schema of a pluggable tree constraint.

3. The **Interface module** performs a bijective relation between events occurring on domain variables (i.e., removals/restorations of values in the domains) and events occurring on the digraph \mathcal{G} (i.e., removals/additions of arcs).

Practically, in the current implementation on the pluggable tree constraint, each time an event occurs on a domain variable, first this event is interpreted by the interface module, next the graph module updates its data structures, next the filtering algorithms dedicated to the tree constraint are applied, and finally the resulting events provided by the constraint module are interpreted in term of domain variable updates.

We are now in position to discuss the interface module. From the domain variables involved in a CSP, basically four kinds of events are distinguished: removal of values in the domains, instantiation of variables, update of lower bounds and update of upper bounds. However, instantiation and bound updates can be easily reduced to a set of removals. Thus, for each kind of event received by the interface from the solver, an event translation to the corresponding set of removals in the graph module is performed. However, all removals are not handled in the same way by the interface.

Indeed, the event at the origin of the considered set of removals is considered to improve the efficiency of the graph module updates. For example, a set of removals related to an instantiation event occurring on a variable leads to a local modification in the neighborhood of the corresponding node in the digraph \mathcal{G} associated with the constraint. This information can be taken into account in order to perform these modifications quite efficiently.

In other words, this new approach for implementing global constraints leads to a reorganization of the code of the constraints. Concerns are clearly separated: data structure management upon domain modifications (events), propagation-related algorithms at the heart of the constraint, and solver/constraint communications. It can be seen as a kind of rationale for global constraints. We think that such a clear separation of concerns is an important point in terms of software engineering.

4 Evaluation

We now report on several experiments we have conducted to evaluate the pluggable tree constraint. First, in Section 4.1, we discuss our experiments on the comparison

between the current *ad-hoc* version of the constraint with the new version proposed in this paper. Then, in Section 4.2, we report on the performance of the pluggable `tree` constraint on two distinct constraint solvers: Gecode and Choco.

All experiments were performed with the Choco constraint solver (version 1.2.04) and Gecode constraint solver (version 1.3.1) on an Intel Xeon CPU with 2.4GHz and a 1GB RAM, but only 128MB were allocated to the Java Virtual Machine.

4.1 Original constraint versus pluggable constraint

Graph Order	Density	Average time (ms)	
		Ad-hoc	Pluggable
25	≤ 0.5	55	45
	> 0.5	90	38
50	≤ 0.5	610	310
	> 0.5	1532	307
75	≤ 0.5	3856	1174
	> 0.5	8896	1064
100	≤ 0.5	13040	3156
	> 0.5	32568	2682
150	≤ 0.5	69220	11543
	> 0.5	219174	9645
200	≤ 0.5	204497	33763
	> 0.5	> 300000	26315

Table 1: Evaluation of the pluggable `tree` constraint with the existing *ad-hoc* implementation.

The aim of these experiments is to show that the pluggable `tree` constraint outperforms the original version implemented within the Choco constraint solver. Moreover, we point out that the dynamic approach proposed throughout this new constraint is, on average, much less sensitive to the variation of the input digraph density which was originally the bottleneck of the previous constraint version [3].

This set of experiments points out two main features of the pluggable `tree` constraint. For each order of graph in $\{25, 50, 75, 100, 150, 200\}$, and the densities in $[0.05; 1]$ with steps of 0.05, we generate 30 instances (i.e. globally, 3600 digraphs). Notice that we add a timeout fixed to 300000ms, and the solver search uses a random variable-value selector.

First, Table 1 highlights a global improvement of the original (ad-hoc) version of the `tree` constraint; indeed, the pluggable version is 3.8 times more efficient in the case of a density less or equal to 0.5, and 10 times more efficient in the case of a density greater than 0.5. Second, Figures 3 and 4 show that the pluggable constraint is significantly more efficient in the case of dense digraphs. Indeed, Figure 3 depicts the behaviors of the pluggable and ad-hoc constraints for a given graph order fixed to 100

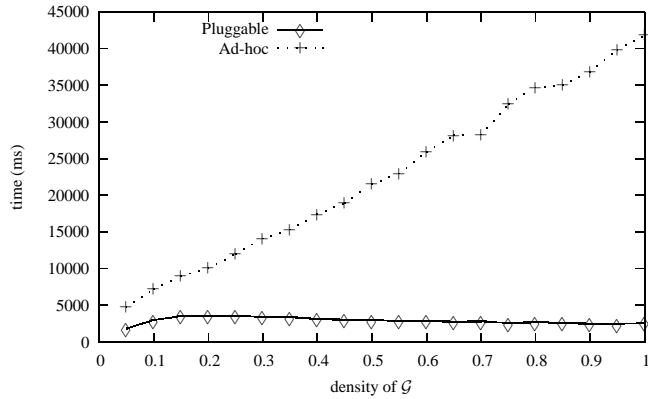


Fig. 3: For a given digraph of order 100, the dotted curve depicts results of the *ad-hoc* implementation of the `tree` constraint while the plain curve depicts the pluggable implementation of the constraint.

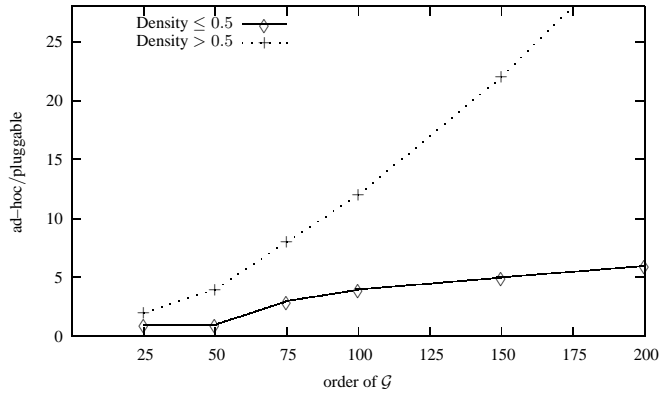


Fig. 4: The dotted curve depicts the ratio between the *ad-hoc* constraint and the pluggable constraint running times, in the case of a density greater than 0.5. The plain curve depicts the same ratio in the case of a density less or equal to 0.5.

nodes. Figure 4 points out the ratio between the pluggable constraint and the *ad-hoc* constraint running times. In both cases, we directly notice that the pluggable version outperforms the original one in the case of dense graphs.

We are now in position to discuss why the original `tree` constraint is outperformed by the new one. First, we detail the feasibility implementation of the constraint. Next, we show how the filtering part was improved. In the following, we denote by n the order of \mathcal{G} and by m the number of arcs in \mathcal{G} .

In [2], a necessary and sufficient condition for the `tree` constraint was introduced: a `tree` constraint has at least one solution iff all sink components of \mathcal{G} contain at least one potential root and $dom(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}] \neq \emptyset$ where, `MINTREE` is the

number of sink components in \mathcal{G} and MAXTREE is the number of potential roots in \mathcal{G} . At each waking up of the constraint, the ad-hoc version of the `tree` constraint computes the strongly connected components (scc) associated with the digraph \mathcal{G} , by a suitable depth first search procedure, introduced by Tarjan [14], running in $O(n + m)$ time. However, computing scc at each waking up of the constraint is useless. Indeed, during the propagation/search steps, removing or adding arcs in \mathcal{G} is a local modification of the digraph then, we can reduce this cost by recomputing scc on a necessary partial graph of \mathcal{G} induced by the scc previously computed. In practice, during search the size of the scc decreases (and the number of scc increases) to reach 1. Thus, dynamically maintaining the scc is part of the improvements provided by the pluggable `tree` constraint.

In the original filtering algorithm proposed in [2], the constraint detected *the strong articulation points*. But in practice, we use a generalization of strong articulation points called dominator nodes [9]. For each dominator nodes, the filtering algorithm detects and removes the outgoing arcs that do not allow to reach at least one potential root. Thus, for each dominator, we have to compute a depth first search tree to detect if a potential root can be reached. Thus, for a given digraph \mathcal{G} , this can be done in $O(nm)$ time. In the pluggable `tree` constraint a new approach is proposed. We associate to the digraph \mathcal{G} , its transitive closure ⁶. Explicitly computing the transitive closure of \mathcal{G} is only done during the initial waking up of the constraint in $O(nm)$. Next, a dynamic handling of the transitive closure is performed by updating the current transitive closure according to a necessary partial graph of \mathcal{G} induced by the events modifying \mathcal{G} . In practice, this dynamic maintain of the transitive closure is very efficient. Finally, the knowledge of the transitive closure provides the reachability conditions that allow us to dynamically filter the dominator nodes when they are identified by the algorithm.

4.2 Towards portability of global constraints

The aim of these experiments is to illustrate that the pluggable `tree` constraint can easily be plugged into several constraint solvers. The choice of the Gecode and Choco constraint solvers leads to show that there is a slight degradation in the case of the Gecode solver due to the interface module.

Gecode is a “propagator-centered” constraint solver. Thus, it cannot dynamically record events occurring on the variables during search. Then, in order to translate the modifications of variable domains in term of arc additions/removals in the graph module, the interface module has to compute events occurring on the domains from the previous awake of the constraint.

The interface module is a global constraint watching domains modifications. At set up, the constraint stores a copy of the domain of each variables. Then, upon each awake, the constraint compares the current variables’ domain with their copy. If a domain has been modified since the last awake, the constraint updates the domain’s copy and sends an arc modification event to the graph module.

⁶ The *transitive closure* of directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is the directed graph $(\mathcal{V}, \mathcal{E}')$ such that for all v, w in \mathcal{V} there is an arc (v, w) in \mathcal{E}' iff there is a non-empty path from v to w in \mathcal{G} .

This computation leads to an overhead of the interface with the Gecode constraint solver. Notice that this is not the case with the Choco interface because Choco is a “variable-centered” constraint solver.

So, the interface module knows what kind of modifications happen on variables domain without having to scan them. Consequently, when the interface module constraint is awoken, it only translates variable events to graph events and send them to the graph module.

For each order of graph in $\{25, 50, 75, 100, 150, 200\}$, and densities in $\{0.05, 0.2, 0.4, 0.5, 0.6, 0.8, 0.95\}$, we generate 50 instances (i.e. globally, 2100 digraphs). Table 2 provides the running time details of each part composing the pluggable `tree` constraint (Figure 2), both for the Choco and Gecode constraint solvers. Notice that both solver searches use the same random variable-value selector.

The column “Interface Module” of Table 2 perfectly points out the overhead due to the interface in Gecode. Moreover, we notice that the running times related to the constraint itself are equivalent in both solvers: the columns “Constraint Module” and “Graph Module” illustrate this result.

Graph Order	Running time <i>Choco</i> (ms)			Running time <i>Gecode</i> (ms)		
	Interface module	Constraint module	Graph module	Interface module	Constraint module	Graph module
25	5	10	27	6	10	27
50	5	57	229	24	56	228
75	11	190	863	58	186	879
100	18	440	2287	120	439	2310
150	50	1466	9524	368	1329	9596
200	82	3214	27551	812	3009	27097

Table 2: Running time comparison of the pluggable `tree` for two distinct constraint solvers (Choco and Gecode) according to the input digraph order. For each one running time of each part Interface, Constraint and Graph is detailed.

4.3 Implementation requirements

A final interesting point about designing pluggable global constraints is related to development times. The original `tree` constraint [2] took something like three months (including the theoretical study of the constraint) to be fully implemented and debugged. The fully dynamic pluggable version took three weeks to be developed (we obviously used our expertise obtained during the first implementation). But, what is more interesting is the time required to port this *choco* specific constraint to the *Gecode* constraint solver: three days!

5 Conclusion and future works

In this paper, we emphasized the interest in decoupling global constraint implementations from the underlying solver. We showed, on a `tree` constraint, that decoupling leads to efficient (compared to the original version of the constraint) and portable global constraint implementations.

As soon as fully incremental algorithms are available for a given global constraint, one should seriously consider developing a solver-independent constraint so as to improve efficiency but also to give the opportunity to the constraint to be widely used. This can be either with other combinatorial problem solving techniques or other constraint solver. Notice that there exist many fully incremental algorithms with low complexity for maintaining individual graph properties. However, combining several of them usually leads to a compromise in choosing the most interesting data structure. In this paper, we made a quite basic choice that certainly would need to be improved.

One key lesson learnt from our paper is that investing in such algorithms is well worth it: efficiency is improved and portability is a reality (it took us three days to port the `tree` constraint to *Gecode* without loss of efficiency).

We are currently working on a generic way to interface global constraints for constraint solvers in order to be able to define a kind of domain specific language for global constraint implementations.

References

1. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Math. Comput. Modelling*, 20(12):97–123, 1994.
2. N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.
3. N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, incompatibility, and degree constraints, with an application to phylogenetic and ordered-path problems. Technical Report 2006-020, Department of Information Technology, Uppsala University, Sweden, April 2006.
4. H. Cambazard and N. Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4):295–313, 2006.
5. D. Eppstein, Z. Galil, and G. Italiano. *Dynamic graph algorithms*. CRC Press, 1997.
6. I. P. Gent, P. Prosser, B. M. Smith, and W. Wei. Supertree construction using constraint programming. In F. Rossi, editor, *Principles and Practice of Constraint Programming CP'03*, volume 2833 of *LNCS*, pages 837–841. Springer-Verlag, 2003.
7. C. Guettier. Solving Planning and Scheduling Problems in Network based Operations. In *Principles and Practice of Constraint Programming CP'07*. LNCS, Springer Verlag, 2007.
8. P. V. Hentenryck and L. Michel. Growing COMET. In F. Benhamou, N. Jussien, and B. O'Sullivan, editors, *Trends in Constraint Programming*, chapter 17, pages 291–297. ISTE, London, UK, May 2007.
9. T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans Program. Lang. Syst.*, 1(1):121–141, 1979.
10. L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *PADL'06*, volume 3819 of *LNCS*, pages 73–87, 2006.

11. J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.
12. P. Schaus, Y. Deville, P. Dupont, and J.-C. Régin. The Deviation Constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07)*, volume 4510 of *LNCS*, 2007.
13. C. Schulte. Comparing Trailing and Copying for Constraint Programming. In D. D. Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, 1999. The MIT Press.
14. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.